CpSc 418 **Homework 1** Due: Sept. 21, 2018, 11:59pm
Early Bird: Sept. 19, 2018, 11:59pm

86 points.

Please submit your solution using the `handin` program. Submit your solution as
cs418 hw1
Your submission should consist of three files:

- `hw1.erl`: Erlang source code for the coding parts your solution.
- `hw1.pdf` PDF for the written response parts of your solution and the plots.
- `hw1_tests.erl`: EUnit tests for your solution.

Templates for `hw1.erl` and `hw1_tests.erl` are available at
`http://www.ugrad.cs.ubc.ca/~cs418/2018-1/hw/1/code.html`.

The tests in `hw1_tests.erl` are not exhaustive. If your code doesn't work with these, it will almost certainly have problems with the test cases used for grading. The actual grading will include other test cases as well.

Please submit code that compiles without errors or warnings. If your code does not compile, we might give you zero points on all of the programming problems. If we fix your code to make it compile, we will take off lots of points for that service. If your code generates compiler warnings, we will take off points for that as well, but not as many as for code that doesn't compile successfully.

We will take off points for code that prints results unless we specifically asked for print-out. For this assignment, the functions you write should return the specified values, but they should not print anything to `stdout`. Using lists:format when debugging is great, but you need to delete or comment-out such calls before submitting your solution. Printing an error message to `stdout` when your function is called with invalid arguments is acceptable but not required. Your code must fail with some kind of error when called with invalid arguments.

1. How long does length take? (**11 points**)

   (a) Measure. (**3 points**)
   Use time_it:t/1 to measure the execution time for the built-in Erlang function length. Test it with lists of length 10000, 20000, 30000, 50000, 100000. For consistent results, please make your measurements on `bowen.ugrad.cs.ubc.ca`. You can try other lengths as well. Report your measurements in a table and as a plot in `hw1.pdf`.

   Hint: include the course Erlang library in your code loading path by giving the command to the Erlang shell:

   ```
   code:add_path("/home/c/cs418/public_html/resources/erl").
   ```

   You can measure the time to execute a function with the function time_it:t/1 from this library. For example:

   ```
   1> X10K = lists:seq(1,10000).  % create a list of 10000 elements
   [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29|...]
   2> time_it:t(fun() -> length(X10K) end).
   [mean,2.0010114937468705e-5,std,2.8817145017420566e-6]
   ```

   reports that the average call took about 20μsec. Note that `time_it:t/1(Fun)` calls the `Fun` repeatedly until about 1 second elapses. More details are given in the documentation.

(b) Think. (**3 points**)
Let N be the length of `List`. Does the execution time for `length(List)` appear to be linear, quadratic, $N \log N$, exponential, or some other function of N? Give a short answer – one sentence is sufficient, and you shouldn't need more than three sentences.

(c) Think. (**3 points**)
Examine the code for `hw1:q1_fun` in `hw1.erl`. Based on your measurements for the execution time of `length`, do you expect the execution time for `hw1:q1_fun(List)` to be linear, quadratic, $N \log N$, exponential, or some other function of N where $N =$ `length(List)`. Give a short answer – you shouldn't need more than three or four sentences.

Hint: Think about this question and answer it. You'll test your answer in question 1d below. You can update your answer if the measurements aren't what you expected.

(d) Measure. (**2 points**)
Measure the run-time for `hw1:q1_fun(List)` for lists of various lengths. You should try lengths where the runtime varies from about `1.0e-5` seconds to `1.0e-2` seconds. The overhead of the time_it:t/1 function is about one microsecond; so times less than 10μsec have a large percentage of overhead. At the other end, $time_it : t/1$ averages over several runs with a total time of about one second. Keeping the run-time to about 0.01 seconds means there will be enough runs to get a reasonably accurate average.

2. Dot-product. (**19 points**)
Let X and Y be vectors of length N. The dot-product of X and Y is

$$\sum_{i=1}^{N} X_i Y_i$$

The template for `hw1.erl` includes the following functions:

- `dot_prod(V1, V2)` is a very "functional" implementation.
- `dot_prod_hr(V1, V2)` is simple, head-recursive version.
- `dot_prod_guard(V1, V2)` is a head-recursive version with a guard.

(a) Measure. (**5 points**)
Measure the execution time of the three versions for dot-product `dot_prod` for lists of various sizes from 1000 to 100,000 elements. For each function, state if the run-time appears to be linear, quadratic, or something else. Give a short justification. Given your answers to question 1, a one sentence answer should suffice for each justification. Report the results in a table. If that doesn't make the answer obvious, include plots in your solution.

(b) Think. (**3 points**)
Does `dot_prod_guard(V1, V2)` call `length(T1X)` and `length(T1Y)` with each recursive call? Give a short justification of your answer.

(c) Tail-recursion. (**5 points**)
Write `dot_prod_tr`, a tail recursive version of `dot_prod`. Yes, I expect you to use a helper function. Add test-cases to `hw1_tests.erl` to test your code. You must add at least three more test cases to get full credit. Check for corner cases.

(d) Measure. (**3 points**)
Measure the execution time of `dot_prod_tr` for the same test-cases that you used for question 2a and report them in a table.

(e) Measure. (**3 points**)

We define the speed-up, S, of program P_{new} relative to program P_{old} as

$$S = \frac{Time(P_{old})}{Time(P_{new})}$$

If $S < 1$, then we have a *slow-down*.

Averaged over your test-cases, which is the fastest implementation of dot product? What is the speed-up compared with the function `dot_prod`? Note: if `dot_prod` *is* the fastest, then the speed-up is 1.

3. Matrix-multiplication.(**10 points**)

Let A be a matrix with n rows and ℓ columns, and let B be a matrix with ℓ rows and m columns. The product of A and B is the matrix C with n rows and m columns where:

$$C_{i,j} = \sum_{k=1}^{\ell} A_{i,k} B_{k,j}$$

where $C_{i,j}$ denotes the entry in the ith row and j^{th} column of C. For example:

$$\begin{bmatrix} 2 & 7 & 3 \\ 1 & 9 & 4 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ -1 & 3 & 2 & -4 \\ 8 & 14 & 9 & 12 \end{bmatrix} = \begin{bmatrix} 19 & 67 & 47 & 16 \\ 24 & 85 & 57 & 16 \end{bmatrix}$$

I'll calculate $C_{2,3}$ in more detail

$$\begin{aligned} C_{2,3} &= \textstyle\sum_{k=1}^{3} A_{2,k} B_{k,3} \\ &= A_{2,1}B_{1,3} + A_{2,2}B_{2,3} + A_{2,3}B_{3,3} \\ &= (1 * 3) + (9 * 2) + (4 * 9) \\ &= 3 + 18 + 36 \\ &= 57 \end{aligned}$$

In Erlang, we will represent a matrix as a list of list, where each row of the matrix is a list. For example, if we let

$$B = \begin{bmatrix} 1 & 2 & 3 & 4 \\ -1 & 3 & 2 & -4 \\ 8 & 14 & 9 & 12 \end{bmatrix}$$

as in the example above, in Erlang we will write

```
B = [ [ 1,  2, 3,  4],
      [-1,  3, 2, -4],
      [ 8, 14, 9, 12] ].
```

Write `matrix_mult/2` function to compute the product of two matrices.

Hints: If $C = AB$, then $C_{i,j}$ is the dot-product of the ith row of A with the jth column of B. Our representation of matrices makes it easy to access the rows of A, but accessing the columns of B is a bit more involved. We write B^T to denote the "transpose" of B where

$$B^T_{i,j} = B_{j,i}$$

In particular, the jth row of B^T corresponds to the jth column of B. I've provided the function `hw1:transpose` that you can use in your implementation of `matrix_mult`.

The matrix representation described here doesn't handle a 0-by-m matrix. You can assume that all matrices have at least one row, and at least one column.

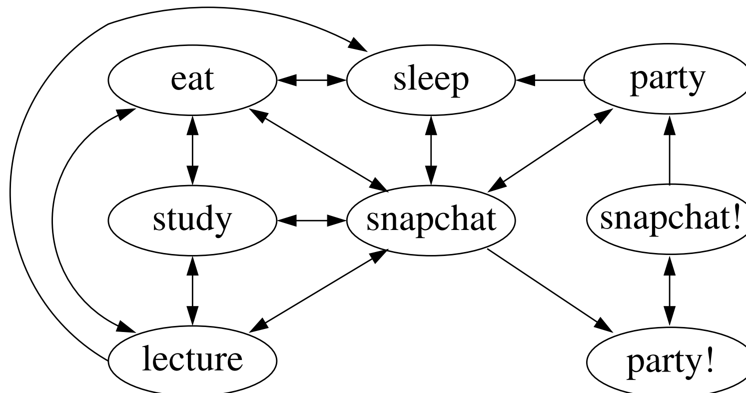

Figure 1: The Life of Brian

	eat	study	lecture	sleep	snap chat	party	snap chat!	party!
eat	0.0	0.4	0.2	0.4	0.2	0.0	0.0	0.0
study	0.3	0.0	0.2	0.0	0.3	0.0	0.0	0.0
lecture	0.2	0.2	0.0	0.0	0.2	0.0	0.0	0.0
sleep	0.1	0.0	0.3	0.0	0.1	0.4	0.0	0.0
snapchat	0.4	0.4	0.3	0.6	0.0	0.6	0.0	0.0
party	0.0	0.0	0.0	0.0	0.1	0.0	0.2	0.0
snapchat!	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
party!	0.0	0.0	0.0	0.0	0.1	0.0	0.8	0.0

Figure 2: The Matrix

4. The Life of Brian.(**20 points**)

Brian is a typical UBC student. His life consists of eating, sleeping, going to lecture, studying, checking in with friends on snapchat, and going to parties – see Figure 1. Once in a while, Brian and his friends go party-hopping, this is indicated by the `party!` and `snapchat!` states in his life.

Being the kind, sympathetic kinds of people that we are in CPSC 418, we want to better understand Brian. One way to do this is with a Markov chain. A Markov chain associates each directed edge in Figure 1 with a probability – for example, if the edge from `lecture` to `study` has weight 0.3, that means there is a 30% probability that Brian will study immediately after that lecture. The sum of the weights of the outgoing edges for a vertex should be 1. We can consider the question of what is the distribution of Brian's activities over a long period of time. We'll just count the activities without worrying about how long he spends at each one. To do this, we represent the graph from Figure 1 as a matrix, M. Element $M_{i,j}$ is the probability that i will be Brian's next after activity j. For example, we could let M be the matrix shown in Figure 2 For example, $M_{1,4} = 0.4$ indicates that after sleeping, Brian will eat as his next activity 40% of the time, and $M_{5,4} = 0.6$ indicates that there is a 60% chance he'll check his snapchat first.

We can represent Brian's random state with a vector. For example, the vector

$$Prob0 \quad = \quad [0.10, 0.35, 0.20, 0.15, 0.09, 0.05, 0.03, 0.03]$$

Indicates that there is a 10% probablility Brian is eating, a 35% probability that he is studying (what a keener!), a 20% chance he is in lecture (perfect attendance), and so on. If we multiply $M\,Prob0$, we

get the probability vector for his next activity:

$$Prob1 = [0.258, 0.097, 0.108, 0.099, 0.36, 0.024, 0.03, 0.024]$$

After one step, there's a 25.8% probability that Brian is eating – all that studying made him hungry.

(a) (**5 points**) The function `markov_steps(M, Prob0)` in hw1.erl computes an update of the probability vector for a single step as described above. Complete the function `markov_steps(M, Prob0, N)` to compute the probabilty vector after `N` steps. In other words, `markov_steps(M, Prob0, N)` should return $M^N\ Prob0$.

(b) (**3 points**) In your `hw1.pdf` file, report the vectors returned by `markov_steps(M, Prob0, N)` for `N` equal to 2, 5, and 10. Round your answers to one place after the decimal point, i.e. 12.3%. Your answers should show that the result of `markov_steps(M, Prob0, N)` is converging toward a limit with larger values of `N`.

(c) Find the fixpoint. (**8 points**)
Under "reasonable" conditions, $M^N\ Prob0$ converges to a limit for any initial probability vector *Prob0*. Complete the function `markov_fixed_point(M, Prob0, ErrTol)` to estimate this limit. We will say that the computation has converged to *Prob* when the largest magnitude element of $(M\,Prob) - Prob$ is less than `ErrTol`.

(d) Test your function. (**4 points**).
Try various vectors for `Prob0`. For example, `[1,0,0,0,0,0,0,0]` represents the case where Brian is initially eating; `[0,0,0,1,0,0,0,0]` represents the case where Brian is initially sleeping; and `[0.125 || _ <- lists:seq(1,8)]` represents the case where Brian has equal probability of starting in any of the eight states from Figure 1. Use `ErrTol = 1.0e-6` for these tests. Report the initial vector and the fixed point in your `hw1.pdf`. What state is most probable at the fixed point and what is its probability? Give your answer for the probabilities to two places after the decimal point, e.g. 12.34%.

5. Will Brian Pass?(**26 points**)
All that partying can be hard on a GPA. We assume:

- Brian fails a term if he goes to ten parties without studying or going to lecture.
- A going to a party is indicated by state 6 (party) or state 8 (party!).
- Studying is indicated by being in state 2 (study), and going to lecture is indicated by being in state 3 (lecture).
- A term has `SleepMax` days. For simplicity, we'll count days by the number of times Brian is in state 4 (sleep). This ignores the possiblity that Brian parties all night and doesn't sleep, or that he falls asleep during a lecture. *C'est la vie.*

While we could devise a Markov Chain to solve this problem, it would have a large number of states and would be too complicated for this assignment – this isn't a probability theory course. Instead, we will simulate a thousand terms or so and determine the percentage of those for which Brian passes.

Function `pass(PartyMax, SleepMax) -> Pass` in `hw1.erl` simulates Brians activity for a term. It uses `hw1:brian_matrix()` as the transition probability matrix, and starts with Brian in state 4 (sleep) – Brian wakes up to start a new term. The simulation should continue until Brian has slept `SleepMax` times without excessive partying (i.e. he passes the term), or when Brian goes to PartyMax parties in a row without studying or going to lecture. In the latter case, Brian fails the term.

To help you implement
`pass(M, PartyMax, SleepMax, CurrentState, PartyCount, SleepCount)`,
I provided function `next_state(M, CurrentState) -> NextState` in `hw1.erl`. `next_state` returns

a random next-state according to the transition probability matrix `M`. If `M` is the matrix from Figure 2 and `CurrentState = 3` (lecture), the `NextState` will be 1 (eat) with probability 20%, 2 (study) with probability 20%, 4 (sleep) with probability 30%, and 5 (snapchat) with probability 30%.

(a) `pass(M, PartyMax, SleepMax, CurrentState, PartyCount, SleepCount)` (8 points)
Implement the function `pass(M, PartyMax, SleepMax, CurrentState, PartyCount, SleepCount)` as described in `hw1.erl`.

`M` is the transition probability matrix.

`PartyMax` if Brian goes to `PartyMax` parties without studying or going to lecture, he fails the term.

`SleepMax` if Brian has slept `SleepMax` terms the term is over. If he hasn't failed yet, Brian passes the term.

`CurrentState` is Brian's current state.

`PartyCount` is a counter that gets incremented each time `CurrentState` is 6 (party) or 8 (party!). `PartyCount` is reset to 0 each time `CurrentState` is 2 (study) or 3 (lecture). If `PartyCount` reaches `PartyMax`, Brian fails the term.

`SleepCount` is a counter that gets incremented each time `CurrentState` is 4 (sleep). If `SleepCount` reaches `SleepMax`, Brian passes the term.

Hints: recursion is your friend. So is `hw1:next_state(M, CurrentState)`.

(b) Try it. (3 points). Execute `hw1:pass(10, 100)` ten times. What percentage of terms did Brian pass?

(c) `pass_count(Ntrials, PartyMax, SleepMax) -> N_Pass.` (5 points)
Running `hw1:pass` enough times to get a good sample is really tedious. Let's have the computer do it for us. Complete `pass_count(Ntrials, PartyMax, SleepMax)`. It should call `pass(PartyMax, SleepMax)` `Ntrials` times and return the number of times that Brian passed.

(d) Try it. (2 points). Execute `hw1:pass_count(100, 10, 100)` ten times, or execute `hw1:pass_count(1000, 10, 100)` once. What percentage of terms did Brian pass?

(e) `pass_worker(ParentPid, Ntrials, PartyMax, SleepMax)` (5 points)
Running a large number of simulations can take a long time, and be very tedious. Parallel computing can help. I've provided most of a parallel implementation for you with the functions

`pass_prob_par(NProcs, Ntrials, PartyMax, SleepMax)` estimate the probability of Brian passing a term using `Ntrials` simulations. These simulations are executed by `NProcs` parallel processors to get better performance.

`pass_spawn(NProcs, Ntrials, PartyMax, SleepMax)` creates `NProcs` worker processes and returns a list of their pids.

`pass_recv(PidList, AccPassed, AccTrials)` receive the number of trials that passed and number of trials attempted from each worker on PidList. Accumulate the results, and return the final ratio.

Your job is to write `pass_worker(ParentPid, Ntrials, PartyMax, SleepMax)`. This function runs `Ntrials` simulations and sends a message to `ParentPid` of the form {WorkerPid, N_Pass, Ntrials} where `WorkerPid` is the pid for this worker process, `N_Pass` is the number of simulations run by this worker for which Brian passed the term, and `Ntrials` is the number of simulations run by this worker.

(f) Try it. (3 points) Measure the time to compute the probability of Brian passing using 10000 simulations and one process:

```
time_it:t(fun() -> hw1:pass_prob(10000, 10, 100) end, 1, [mean, values]).
```

The parameter `1` for `time_it:t` says to run one trial, and the parameter `[mean, values]` says to return the average run time, and the values returned by the function being tested. Now make measurements for the parallel version with 2, 4, 8, 16, 32, 64, and 128 processes. For example, you could give the command:

```
time_it:t(fun() -> hw1:pass_prob_par(2, 10000, 10, 100) end, 1, [mean, values]).
```

Report the run-times in `hw1.pdf`. What number of processes achieves the greatest speed-up. What is this speed-up?

Why?

This is a follow-up to PIKA 1 to make sure that everyone is confident programming in Erlang. Often, people who are new to functional programming find that they miss their loops. The questions in this assignment require various forms of traversing data structures, obviously without loops. They are arranged roughly in an order from simple programming patterns to some more general ones. These questions also introduce examples such as matrix multiplication that we will revisit later in the term.

I have found that reading code is an important part of learning to program well. For this assignment, you read more code than you write. You'll need to read the code to understand what you need to write.

Question 1: A theme throughout this course is to analyze and measure. This is a simple problem to demonstrate this approach.

Question 2: When we use CUDA in the second half of the term, we'll find that GPUs are especially good at numerical computation. Dot-products are a simple example. This problem explores four implementations of dot product and the performance trade-offs. I found it to be a fun problem because I was surprised by one of the answers.

Question 3: Matrix multiplication is a very common operation in numerical computing, and it is very well suited for parallel implementation. Everyone should have seen matrix multiplication in their linear algebra (a.k.a. matrix algebra) course, but I thought this is a good refresher.

Question 4: Now it's time to have some fun with the `matrix_mult` function. Markov chains occur in many contexts: modeling internet server workloads, wait-times at grocery stores or hospitals, traffic congestion patterns, modeling epidemics, etc. One of the best known (for computer scientists) is Google's page-rank algorithm. For page rank, Brian randomly clicks on links on web pages. Occasionally, Brian goes to a completely random web page. Using the same kind of computation that we did here, Google and other search engines, find the steady-state distribution for visiting web pages. The pages that are more likely in this distribution are often the most relevant. Google, Bing, DuckDuckGo and others have all made their refinements on this approach, but this is the basic idea.

Question 5: This is an example of a Monte Carlo simulation problem. Monte Carlo simulation is an example of an "Embarrassingly Parallel" problem. They are also very important. In an earlier term, I was contacted by a local company looking for programmers who could write parallel code for Monte Carlo stress tests of investment portfolios that banks are required to run.

The Library, Errors, Guards, and other good stuff

The CPSC 418 Erlang Library: your code *must* run on the CS department linux machines. The time_it:t/1 function used in this assignment is from the course library. To access this library from the CS department machines, give the following command in the Erlang shell:

```
3> code:add_path("/home/c/cs418/public_html/resources/erl").
```

You can also set the path from the command line when you start Erlang. I've included the following in my `.bashrc` so that I don't have to set the code path manually each time I start Erlang:

```
function erl {
  /usr/local/bin/erl erl -eval 'code:add_path("/home/c/cs418/public_html/resources/erl")' "$@"
}
```

See http://erlang.org/doc/man/erl.html for a more detailed description of the `erl` command and the options it takes.

If you are running Erlang on your own computer, you can get a copy of the course library from
http://www.ugrad.cs.ubc.ca/~cs418/resources/erl/erl.tgz
Unpack it in a directory of your choice, and use code:add_path as described above to use it. Changes may be made to the library to add features or fix bugs as the term progresses. I try to minimize the disruption and will announce any such changes.

Compiler Errors: if your code doesn't compile, it is likely that you will get a zero on the assignment. Please do not submit code that does not compile successfully. After grading all assignments that compile successfully, we *might* look at some of the ones that don't. This is entirely up to the discretion of the instructors and TAs. If you have half-written code that doesn't compile, please comment it out or delete it.

Compiler Warnings: your code should compile without warnings. In my experience, most of the Erlang compiler warnings point to real problems. For example, if the compiler complains about an unused variable, that often means I made a typo later in the function and referred to the wrong variable, and ended up not using the one I wanted. Of course, the "base case" in recursive function often has unused parameters – use a `_` to mark these as unused. Other warnings such as functions that are defined but not used, the wrong number of arguments to an io:format call, etc., generally point to real mistakes in the code. We will take off points for compiler warnings.

Printing to stdout: please don't unless we specifically ask you to. If you include a *short* error message when throwing an error, that's fine, but not required. If you print *anything* for a case with normal execution when no printing was specified, we will take off points.

Guards: in general, guards are a good idea. If you use guards, then your code will tend to fail close to the actual error, and that makes debugging easier. Guards also make your intentions and assumptions part of the code. Documenting your assumptions in this way makes it much easier if someone else needs to work with your code, or if you need to work with your code a few months or a few years after you originally wrote it. There are some cases where adding guards would cause the code to run much slower. In those cases, it can be reasonable to use comments instead of guards. Here are a few rules for adding guards:

- If you need the guard to write easy-to-read patterns, use the guard.
- If adding the guard makes your code easier to read (and doesn't have a significant run-time penalty), use the guard.
- If a function is an "entry point" into your code (e.g. and exported function) it's good to have your assumptions about arguments clearly stated. If you can do this with guards, that is great.
- Adding lots of little guards to every helper function can clutter your code. Write the code that you would want others to write if you are going to read it.
- In some cases, guards can cause a severe performance penalty. In that case, it's better to use a wrapper function so you can test the guards once and then go on from there, or to use comments. Comment don't slow down the code.

A common case for omitting guards occurs with tail-recursive functions. We often write a wrapper function that initializes the "accumulator" and then calls the tail-recursive code. We export the wrapper, but the tail-recursive part is not exported because the user doesn't need to know the details of the tail-recursive implementation. In this case, it makes sense to declare the guards for the wrapper function. If

those guarantee the guards for the tail-recursive code, and the tail recursive code can only be called from inside its module, then we can omit the guards for the tail-recursive version. This way, the guards get checked once, but hold for all of the recursive calls. Doing this gives us the robustness of guard checking **and** the speed of tail recursion.

Unless otherwise noted or cited, the questions and other material in this homework problem set is copyright 2018 by Mark Greenstreet and are made available under the terms of the Creative Commons Attribution 4.0 International license `http://creativecommons.org/licenses/by/4.0/`